Informe TP2

Garcia Conejero, Lautaro
Howlin, Simon Pedro

¿Cómo compilar el archivo?

Este trabajo utiliza **CMake** como sistema de compilación. Por lo tanto, para compilar y ejecutar los ejercicios sólo es necesario seguir estos sencillos pasos desde la terminal:

Ejemplo: Compilación del Ejercicio 1

Ingresar a la carpeta del ejercicio:

cd Ej_1

Entrar a la carpeta build:

cd build

Generar los archivos de compilación y compilar el proyecto:

cmake .. && make

Ejecutar el programa:

./bin/Ej1

Importante:

Para compilar los ejercicios 2 y 3 es **requisito** utilizar el estándar **C++20**.

Aclaraciones:

- Las carpetas build y data no eran posibles guardarlas vacías ya que estas contienen los archivos ejecutables. Es por esto que se genera un archivo .gitkeep para que se guarden dentro del github.

Ejercicio 1

1 Clases Implementadas

1.1 Clase Pokemon

Esta clase representa a un Pokémon específico. Sus atributos principales son:

- nombre (std::string): Identificador único del Pokémon.
- experiencia (int): Nivel de experiencia acumulado por el Pokémon.

Se implementaron los siguientes métodos y operadores:

- Constructor con validación para asegurar que la experiencia nunca sea negativa, garantizando la integridad del dato.
- getNombre() y getExperiencia(): Métodos getter que proporcionan acceso controlado y encapsulado a los atributos privados.
- Operador ==: Permite comparar dos Pokémon por nombre, esencial para la unicidad en el sistema y la posibilidad de verificar la existencia de un Pokemon dentro de la Pokédex.
- Operador <<: Facilita la impresión amigable del Pokémon, mostrando claramente sus atributos por consola.
- serializar() y deserializar(): Métodos cruciales que permiten guardar y recuperar información desde archivos binarios, garantizando persistencia de datos. Como sus datos son atributos simples, la serialización y deserialización se concentra en solo ingresar los valores respetando los órdenes.

Clase PokemonHash: Functor que implementa un hash basado exclusivamente en el nombre, necesario para usar Pokémon como clave en un contenedor tipo unordered_map. Permite que el “hasheo” considere únicamente el nombre del pokémon.

1.2 Clase PokemonInfo

Esta clase almacena información detallada y específica del Pokémon:

- tipo (string): Tipo elemental del Pokémon.
- descripcion (std::string): Breve descripción que detalla características distintivas del Pokémon.
- AtaquesPorNivel (vector<pair<string, int>>): Lista ordenada que guarda ataques disponibles, acompañados del daño que producen.
- ExperienciaPorNivel (vector): Lista que define claramente la experiencia necesaria para avanzar en cada nivel del Pokémon.

Justificación detallada del uso de contenedores

- **vector<pair<string, int>>:** Este contenedor fue seleccionado específicamente para almacenar la lista de ataques debido a que mantiene un orden claramente definido. La estructura pair<string, int> permite asociar directamente el nombre de cada ataque con su respectivo daño, simplificando el acceso y la gestión de estos datos. Además, los vectores ofrecen una alta eficiencia en el recorrido secuencial, que es la operación predominante al mostrar los ataques disponibles para cada Pokémon. Esta elección asegura que la impresión, modificación o acceso a ataques específicos se realice de manera eficiente y ordenada. Otra opción útil es usar un map que contenga el nombre como clave y el daño como valor. Como no es tan grande los datos a guardar, usar el vector no debería afectar al tiempo que tarda el programa.
- **vector<int>:** Para almacenar la experiencia requerida por cada nivel, se optó por utilizar un vector<int> debido a la necesidad de acceder rápidamente a los datos mediante índices específicos. Los niveles de experiencia son fijos y su posición está claramente definida (por ejemplo, nivel 1, 2 y 3). Esta característica hace que el vector sea ideal, proporcionando acceso directo y eficiente (complejidad O(1)) para consultar o actualizar la experiencia necesaria para alcanzar un nivel concreto.

Métodos implementados:

- Getters para cada atributo, promoviendo encapsulación.

- Operador << que permite impresión clara y estructurada de toda la información.
- Los métodos serializar() y deserializar() son fundamentales para la gestión efectiva y persistente de la información. En este caso, debido a que las clases contienen múltiples contenedores (como vectores de pares y vectores de enteros), fue necesario ampliar la lógica de serialización para no solo guardar los datos, sino también el tamaño de cada contenedor. Esto es crucial porque, al deserializar, se requiere conocer cuántos elementos hay para poder reconstruir correctamente las estructuras de datos en memoria. De esta forma, se asegura que la información se guarde y recupere íntegramente y sin pérdida, manteniendo la integridad de los objetos.

1.3 Clase Pokedex

La clase Pokedex es el núcleo del sistema, usando internamente:

unordered_map<Pokemon, PokemonInfo, PokemonHash> pokemons;

Este contenedor permite una rápida recuperación de información debido al uso de hashing personalizado.

Funciones detalladas:

- sumarPokemon: Inserta un Pokémon solo si no existe previamente, evitando duplicados. Si el archivo de data está activado, actualiza automáticamente los datos guardados.
- existePokemon: Busca en pokemons si existe el pokemon pasado como parámetro. se reutiliza esta función en mostrarPokemon.
- mostrarPokemon: Busca y presenta detalladamente la información de un Pokémon específico, avisa claramente si el Pokémon no está registrado.
- mostrarTodos: Realiza un recorrido completo del contenedor e imprime la información de todos los Pokémon almacenados.
- guardarEnArchivo y cargarDesdeArchivo: Manejan eficazmente la persistencia de los datos mediante archivos binarios. serializar y deserializar los datos respetando el orden y usando los métodos de las clases Pokemon y PokemonInfo.

El constructor adicional permite especificar un archivo para cargar automáticamente la información al inicio del sistema, proporcionando comodidad y eficiencia.

2. Casos de Prueba

Para validar integralmente el sistema se añadieron tres Pokémon específicos según el enunciado:

Squirtle (Tipo Agua, XP: 100): Ataques Pistola Agua, Hidrobomba, Danza Lluvia; niveles experiencia 0, 400, 1000.

Bulbasaur (Tipo Planta, XP: 270): Ataques Latigazo, Hoja Afilada, Rayo Solar; niveles experiencia 0, 300, 1100.

Charmander (Tipo Fuego, XP: 633): Ataques Ascuas, Lanzallamas, Giro Fuego; niveles experiencia 0, 250, 1300.

Se realizaron múltiples pruebas para comprobar:

- mostrarTodos(): Verificó que todos los Pokémon sean mostrados correctamente.
- mostrarPokemon("Squirtle"): Mostró correctamente la información, ya que la búsqueda se basa solo en el nombre.
- mostrarPokemon("Pikachu"): Confirmó la respuesta clara y adecuada cuando el Pokémon no está registrado.

Se agregó También una segunda prueba que comprueba la funcionalidad de la serialización:

- Comprueba que si el archivo no fue cargado, la pokédex está vacía
- Si ya hay pokemones cargados, la Pokédex mostrará los datos cargados

3. Serialización y Persistencia

La funcionalidad opcional de persistencia fue implementada mediante:

- La serialización binaria de los objetos, permitiendo que la información persista tras la finalización del programa.
- Actualización automática del archivo tras cada nueva inserción de Pokémon, garantizando datos siempre actualizados.
- Esto asegura que la información nunca se pierda y pueda ser recuperada fácilmente.

Ejercicio 2

El segundo ejercicio del trabajo práctico consiste en desarrollar una simulación del despegue coordinado de cinco drones cuadricópteros ubicados en círculo dentro de un hangar. Debido a que los drones generan turbulencia, cada uno debe asegurarse de que las dos zonas adyacentes (a su izquierda y derecha) estén libres antes de despegar, para evitar interferencias y garantizar un vuelo estable.

1. Problema a Resolver

Cinco drones están ubicados en círculo, numerados del 1 al 5. Entre cada par de drones vecinos hay una zona de turbulencia (interferencia). En total, hay 5 zonas, numeradas del 1 al 5, cada una entre dos drones consecutivos.

- El drone i necesita asegurar que las zonas i y i+1 estén libres para despegar.
- La zona 6 es equivalente a la 1 por el carácter circular.

2. Metodos de resolucion

2.1 Recursos compartidos

- Un vector global de std::mutex llamado z_turb con 6 elementos (índice 0 reservado para el mutex de consola).
- Cada mutex representa una zona de turbulencia que puede estar libre u ocupada.

2.2 Bloqueos de los mutex

- Cada drone debe adquirir simultáneamente los mutex de sus dos zonas laterales antes de despegar.
- Se utiliza scoped_lock para bloquear ambas zonas a la vez, evitando condiciones de carrera y deadlocks.
- Se bloquea primero la zona con menor índice y luego la mayor, garantizando un orden fijo.
- Un mutex especial, z_turb[0], protege las impresiones para evitar que los mensajes de diferentes hilos se mezclen en la salida estándar.

3. Implementación en C

3.2 Clase SimDespegue

Esta función representa el proceso que sigue cada dron para despegar de manera segura, garantizando la exclusión mutua en las zonas de turbulencia adyacentes y la correcta sincronización de mensajes en pantalla.

Paso 1: Mensaje de espera

- Para comunicar que un dron está esperando su turno para despegar, se bloquea un mutex especial que controla el acceso exclusivo a la consola de salida. Esto se hace llamando a lock() sobre z_turb[0], que es el mutex reservado para esta tarea.

Paso 2: Caso especial para un solo dron

- Si el sistema tiene configurado que solo hay un dron (Drones == 1), no se requiere realizar ningún bloqueo adicional para las zonas laterales porque no existe interferencia posible. En este caso, el dron imprime directamente el mensaje de despegue, realiza la simulación de vuelo y luego indica que alcanzó la altura segura de 10 metros. Este caso simplifica la ejecución y evita bloqueos innecesarios.

Paso 3: Bloqueo simultáneo de zonas laterales

- Para los casos con más de un dron, la función utiliza un objeto scoped_lock para bloquear simultáneamente los dos mutexes que representan las zonas de turbulencia adyacentes al dron actual.
- Este mecanismo bloquea ambos mutexes en forma atómica y en un orden predefinido, eliminando la posibilidad de deadlocks. Así, el dron se asegura de tener el control exclusivo de ambas zonas necesarias antes de iniciar el despegue.
- De acá muestra por terminal los mensajes que

4. Justificación del diseño y sincronización

- **Uso de scoped_lock para múltiples mutexes:**
  Garantiza la adquisición atómica de ambos mutexes de zonas laterales, eliminando la posibilidad de deadlocks por adquisición en orden distinto.
- **Mutex exclusivo para consola:**
  El uso de z_turb[0] para sincronizar las impresiones evita la mezcla de mensajes, logrando una salida clara y secuencial.
- **Uso de jthread:**
  Facilita el manejo automático del ciclo de vida de los hilos, sin necesidad de llamadas explícitas a join() o detach(), mejorando la seguridad del código.
- **Simulación temporal con sleep_for:**

  Se respeta la restricción de no usar esperas activas para sincronización, utilizando únicamente la espera para simular el despegue real.

Ejercicio 3

1. Problema a Resolver

Este ejercicio consiste en simular un sistema concurrente compuesto por sensores y robots en una planta industrial automatizada. Los sensores generan tareas periódicas de inspección que deben almacenarse en una cola centralizada y compartida. Los robots consumen esas tareas para realizar acciones correctivas o de mantenimiento.

El sistema debe garantizar que:

- Los sensores puedan generar tareas simultáneamente sin interferencias.
- Los robots procesen las tareas en orden FIFO, esperando cuando la cola esté vacía.
- Los robots finalicen correctamente su trabajo una vez que todos los sensores hayan terminado y se hayan procesado todas las tareas.
- Se eviten condiciones de carrera y corrupciones en la cola compartida.
- La salida por consola se muestre clara y ordenadamente, sin mezclas de mensajes.

2. Métodos de Resolución

2.1 Recursos Compartidos

- **Cola de tareas** (queue<Tarea>): Almacena las tareas pendientes para procesar.
- **Mutex único** (m_tarea): Protege la cola y variables compartidas (sensorCompleto, tareasFinalizadas) para evitar accesos concurrentes no controlados.
- **Variable de condición** (cv): Permite a los robots esperar cuando no hay tareas y recibir notificaciones cuando se agregan nuevas o cuando los sensores terminan.

2.2 Producción y Consumo

- Los **sensores** son hilos productores que generan un número aleatorio de tareas (entre 1 y 5), y las añaden a la cola bajo protección mutex.
- Los **robots** son hilos consumidores que extraen tareas de la cola en orden, procesándolas con una simulación de tiempo.

- Se usan mecanismos de sincronización para coordinar el acceso y notificación, evitando race conditions y garantizando un procesamiento ordenado.

3. Implementación en C++

3.1 Definición de la tarea

Cada tarea es representada por una estructura Tarea que contiene:

- IdSensor: identificador del sensor que generó la tarea.
- IdTarea: identificador único de la tarea.
- Descripcion: texto descriptivo que incluye el número de tarea.

3.2 Función generar_tarea(int idSensor)

Esta función es ejecutada por cada hilo sensor y realiza:

- Genera un número aleatorio de tareas entre 1 y 5.
- **Para cada tarea:**
    - Bloquea el mutex m_tarea para proteger la cola.
    - Crea un nuevo objeto Tarea y lo inserta en la cola.
    - Imprime un mensaje indicando qué tarea fue generada.
    - Notifica a todos los robots que hay tareas disponibles con cv.notify_all().
    - Desbloquea el mutex.
    - Duerme 175 ms simulando el tiempo de generación.

- **Al finalizar la generación de todas sus tareas:**
    - Bloquea nuevamente el mutex.
    - Incrementa sensorCompleto.
    - Si todos los sensores terminaron (sensorCompleto == sensoresCantidad), marca tareasFinalizadas = true y notifica a todos con cv.notify_all().

3.3 Función procesar_tarea(int idRobot)

Ejecutada por cada hilo robot, esta función sigue el siguiente ciclo:

- **Mientras el sistema esté activo:**
    - Bloquea el mutex m_tarea.
    - Espera en cv hasta que haya tareas disponibles o se haya terminado la generación (tareasFinalizadas).
    - Si la cola está vacía y no se generarán más tareas, rompe el ciclo y finaliza.
    - Si hay tareas:
        - Extrae la primera tarea de la cola.
        - Imprime un mensaje indicando la tarea procesada.
    - Desbloquea el mutex.
    - Duerme 250 ms simulando el procesamiento.

4. Justificación del diseño y sincronización

- **Mutex único para recursos compartidos**:
  Se usa un solo mutex (m_tarea) para proteger la cola y las variables compartidas, garantizando que sólo un hilo acceda a la cola en cada momento.
- **Variable de condición para coordinación**:
  Permite a los robots esperar bloqueados cuando no hay tareas, y despertar eficientemente cuando se agregan nuevas tareas o cuando todos los sensores han terminado.
- **Uso de unique_lock y condition_variable::wait**:
  Proporciona manejo correcto y seguro del mutex durante la espera, liberándolo mientras espera y recogiéndolo al despertar.
- **Notificación a todos los consumidores**:
  Se emplea notify_all() para garantizar que todos los robots se despierten para revisar si hay tareas disponibles o si deben terminar.
- **Simulación temporal con sleep_for**:
  Respetando la consigna, los tiempos de generación y procesamiento se simulan con pausas precisas, sin emplear esperas activas.